Proceedings of GT2009
ASME Turbo Expo 2009: Power for Land, Sea and Air
June 8-12, 2009, Orlando, USA

GT2009-60052

AN ACCELERATED 3D NAVIER-STOKES SOLVER FOR FLOWS IN TURBOMACHINES

Tobias Brandvik and Graham Pullan
Whittle Laboratory
Department of Engineering
University of Cambridge
Cambridge, UK
tb302@cam.ac.uk, gp10006@cam.ac.uk

ABSTRACT

A new three-dimensional Navier-Stokes solver for flows in turbomachines has been developed. The new solver is based on the latest version of the Denton codes, but has been implemented to run on Graphics Processing Units (GPUs) instead of the traditional Central Processing Unit (CPU). The change in processor enables an order-of-magnitude reduction in run-time due to the higher performance of the GPU. Scaling results for a 16 node GPU cluster are also presented, showing almost linear scaling for typical turbomachinery cases. For validation purposes, a test case consisting of a three-stage turbine with complete hub and casing leakage paths is described. Good agreement is obtained with previously published experimental results. The simulation runs in less than 10 minutes on a cluster with four GPUs.

NOMENCLATURE

$\mathbf{u}$	Velocity vector
λ	Thermal conductivity
ρ	Density
τ	Viscous stress tensor
c_v	Specific heat at constant volume
R	Gas constant
e	Specific entergy $= c_v T + \frac{1}{2}v^2$
h_0	Stagnation enthalpy
p	Pressure
p_0	Stagnation pressure
T	Temperature
t	Time
v	Velocity magnitude
Δs	Entropy change

INTRODUCTION

A key metric in the evaluation of a CFD solver is the time taken per node per timestep. Advances in hardware and, to a lesser extent, algorithms have enabled this metric to fall continuously. In terms of low-cost commodity hardware (CPUs), processor clock speed and changes in processor architecture were the primary drivers for this advance in the 1980s and 1990s. At present, parallel computing using clusters of multi-core processors is the key enabler. None-the-less, a step change in this metric (at least one order of magnitude) would be invaluable in bringing high fidelity (LES, DNS) solutions into routine industrial use or in making the current standard of design tools interactive.

Alongside the shift from single- to multi-core CPUs that has occurred over the last five years, advances have also been made for other types of processors. There are now a variety of chips on the market that exhibit a higher level of parallelism (and hence performance) than CPUs, including Graphics Processing Units (GPUs) and the STI Cell microprocessor. Taking advantage of such processors for CFD calculations could deliver a step change in performance today, but requires significant changes to the underlying code.

There has been only limited work reported so far on the use

of these novel processors for CFD. The present authors have presented results for 2D [1] and 3D [2] Euler solvers for turbomachinery applications, achieving speed-ups of an order of magnitude for both AMD and NVIDIA GPUs compared to a single CPU core. Similar results, with the extension to include a full multigrid scheme, have also been presented by Elsen et al. [3] for a 3D Euler solver with applications to external flows.

In this paper, a new three-dimensional Navier-Stokes solver which runs on GPUs is presented. The solver, called Turbostream, includes a mixing-length turbulence model and the capability of simulating both steady and unsteady flows in multi-row turbomachines. In addition, the solver is able, through the use of the Message Passing Interface (MPI), to utilise many GPUs together to solve problems that are beyond the scope of a single processor. To the authors' knowledge, this represents the first viscous solver for engineering flows running on a large number of GPUs.

We first present an overview of many-core processors, focusing specifically on GPUs and how they relate to CPUs. A brief description of the solver algorithm and implementation is then given, followed by a discussion of the solver's performance as it compares to an older CPU solver that implements the same algorithm. Finally, a test case consisting of a three-stage turbine with hub and casing leakage paths is presented. Comparisons showing good agreement with experiments are also given.

MANY-CORE PROCESSORS

In order to keep increasing the computational power of their chips, the semiconductor industry has now moved from single- to multi-core designs. The reason for this switch is the requirement to keep within an acceptable power envelope of around 100-200 Watts. Given this constraint, Borkar [4] has identified the diminishing returns of core complexity as the main motivator for multi-core processors. He refers to this relationship as Pollack's rule, which states that the computational power of a core is roughly proportional to the square root of its complexity. Therefore, it is clearly more power-efficient to use the extra transistors offered by technology scaling to add more cores to a chip, rather than to increase the complexity of the exisiting ones. This trend is widely expected to continue, resulting in 1000-core chips being common-place within the next 10 years.

For software development, the consequenses of this explosion in core count are far-reaching, requiring significant changes to old codes and the algorithms they use. In this paper, the problem is approached from the point of view of a developer rewriting an existing Fortran structured grid CFD solver. To set the stage, we first introduce the two different processors that will be considered. The first is a quad-core Intel Xeon CPU which is representative of the processors that run most CFD solvers today; the second is the less familiar design of an NVIDIA GPU that is the current target processor of Turbostream. A schematic overview of both processors can be seen in Fig. 1.

Intel Xeon

The processor considered here is the 2.33GHz Harpertown variant in Intel's Xeon line. It is a general-purpose processor, meaning that it is capable of running an operating system on its own. Harpertown is a dual-die quad-core, i.e. two dual-cores put together in the same package. Each core has its own 32 KB L1 cache, while each die has its own 4 MB L2 cache shared between the two cores. Each core also has an adder and a multiplier for 128-bit vectors, making it capable of 18.6 single precision GFLOP/s (10^9 floating point operations per second). The theoretical aggregate performance of the whole chip is therefore 74.4 GFLOP/s. The cores have access to external memory through the front side bus (FSB) at a rate of 10.6 GB/s.

The programming approach used for structured grid applications on CPUs is now well established. Typically, some form of domain decompositioning with ghost cells is used to split the domain between the processor cores. One CPU process is then started per core. Each process iterates over its part of the domain, updating the grid variables as it goes along. At the end of each iteration, the processes exchange data with each other to update the ghost cells by using MPI. A variation on this approach is to only start one CPU process per processor, and then parallelise the work given to that processor across its cores using threads (either created explicitly or through compiler directives such as OpenMP). However, since most established codes were already parallelised with MPI before the arrival of multi-core CPUs, the pure MPI approach involves less work and seems to be more popular.

NVIDIA GPU

The NVIDIA GPU considered here is the latest GT200 chip. The older G80 chip is also used in some of the performance measurements presented later - this has approximately half the performance of the GT200.

The GT200 is is designed to accelerate the rendering of 3D scenes in computer games, so its volume sales are driven by the computer games industry. Unlike a CPU, it is not a general-purpose chip and cannot run an operating system. Instead, it is sold as part of an add-in card (graphics card) that comes with its own on-board memory and plugs into an expansion slot on the PCI-Express bus. The GT200 consists of 30 multi-processors (MP), each of which contains 8 scalar processing units (SP) and 16 KB of explicitly managed local storage (referred to as shared memory). Each MP has its own instruction counter and operates independently of the others. Each SP can schedule one multiply and one multiply-add operation (both single precision) per cycle, giving a theoretical peak performance of 933GFLOP/s at 1.296GHz. By using a wide 512-bit bus to the graphics card's on-board GDDR3 memory, the GT200 achieves a maximum band-

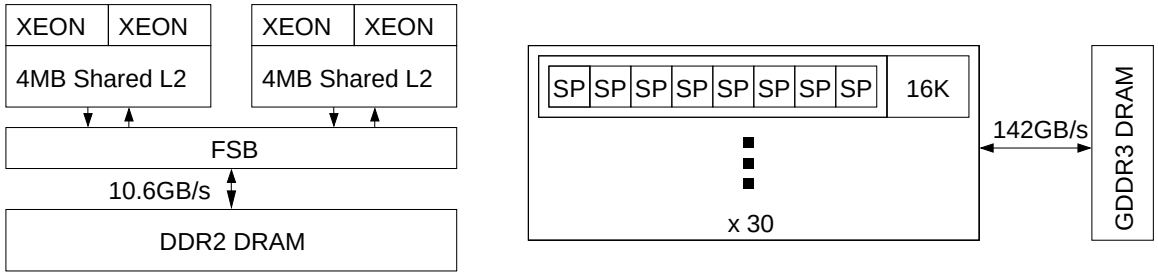

Figure 1. CPU and GPU architecture overview

width of 141.7GB/s.

To simplify the programming of their GPUs, NVIDIA has developed an extension to the C programming language called CUDA. In a CUDA program, the developer sets up a large number of threads (often several thousand) that are grouped into thread blocks. A CUDA thread is the smallest unit of execution and has a set of registers and a program counter associated with it. This is similar to traditional CPU threads, but CUDA threads are much less expensive to create and swap between. Each thread block is executed on a single multi-processor. It is possible to synchronize the threads within a block, allowing the threads to share data through the shared memory. Given that a thread block can consist of more threads than the number of processors in a multi-processor, the hardware is responsible for scheduling the threads. This allows it to hide the latency of fetches from the on-board memory by letting some threads perform computations while others wait for data to arrive. For structured grid applications, a natural way of organising the code is to split the main grid into smaller grids which can fit into the shared memory. A block of threads is then started within each of the smaller grids to compute the updated variables.

ALGORITHM

Turbostream is heavily based on the long line of codes from Denton. In particular, it uses the same algorithm as that of the latest Denton code, TBLOCK, with only minor differences in the way that this algorithm is implemented. A complete description of TBLOCK is given by Klostermeier [5], while shorter overviews and examples of its application to turbomachinery problems have been published by Reid et al. [6] and Rosic et al. [7]. In addition, the motivation for the current method can be traced through a series of papers by Denton [8–11]. Here, we give a basic overview of the algorithm as it is used in Turbostream.

Algorithm overview

Turbostream uses a multi-block topology with arbitrary patch interfaces to capture complex geometries. Information is passed between blocks using surface patches which contain nodes that are physically coincident but reside on different blocks. The flow properties at these nodes are averaged at the end of each timestep. Parallel simulations on a cluster of processors can be performed by decomposing the domain on a block basis. This decomposition is performed as an automatic pre-processing step in which each block can be further split into smaller blocks to achieve better load-balancing.

The Navier-Stokes equations are discretised using a finite volume method with vertex storage in a structured grid of hexahedral cells. In this technique, the equations in their integral form for mass, momentum and energy are used:

Mass:

$$\frac{\partial}{\partial t}\int_{\Omega}\rho \mathrm{d}\Omega + \oint_{\mathrm{A}}\rho\mathbf{u}.\mathrm{d}\mathbf{A} = 0 \quad (1)$$

Momentum:

$$\frac{\partial}{\partial t}\int_{\Omega}\rho\mathbf{u}\mathrm{d}\Omega + \oint_{\mathrm{A}}\rho\mathbf{u}(\mathbf{u}.\mathrm{d}\mathbf{A}) + \oint_{\mathrm{A}}p\mathrm{d}\mathbf{A} - \oint_{\mathrm{A}}\tau.\mathrm{d}\mathbf{A} = 0 \quad (2)$$

Energy:

$$\frac{\partial}{\partial t}\int_{\Omega}\rho e\mathrm{d}\Omega + \oint_{\mathrm{A}}\rho h_0\mathbf{u}.\mathrm{d}\mathbf{A} - \oint_{\mathrm{A}}(\tau.\mathbf{u}).\mathrm{d}\mathbf{A} - \oint_{\mathrm{A}}\lambda\nabla T.\mathrm{d}\mathbf{A} = 0 \quad (3)$$

where Ω is a control volume bounded by a surface A. The above integrals are performed on each cell in the grid using a second-order spatial discretisation.

The equations can be expressed less formally as

$$\frac{\partial\mathbf{U}}{\partial t} = \frac{\Sigma\mathbf{F}}{V} + \mathbf{S} \quad (4)$$

where **U** is a vector containing the primary flow variables, **F** is a vector containing the fluxes of the primary variables, the flux summation is over the faces of the cell, **S** contains any source terms and V is the volume of the cell. In the manner described by Denton [10], the source vector is here used to hold the viscous terms. This equation is integrated forward to reach a steady state ($\frac{d\mathbf{U}}{dt} \approx 0$) using the Scree scheme (see Denton [12]):

$$\Delta\mathbf{U} = \left(2\frac{\partial\mathbf{U}}{\partial t}\Big|_n - \frac{\partial\mathbf{U}}{\partial t}\Big|_{n-1}\right)\Delta t \tag{5}$$

where the subscripts refer to the time step that the derivatives were evaluated at.

Since the integrals are evaluated for a hexahedral cell and the solver uses vertex storage, the cell-based $\Delta\mathbf{U}$ has to be distributed to the surrounding vertices, each receiving an eighth. Finally, to maintain numerical stability, artifical smoothing is then applied to all the flow variables. Presently, only second-order smoothing is used, but a more traditional blended second- and fourth-order smoothing procedure is being considered.

Turbulence model

The effect of turbulence is modelled using a simple algebraic mixing-length model in which the turbulent viscosity ν_t is related to a length scale l_{mix} over which turbulent mixing is assumed to take place:

$$\nu_t = l_{mix}^2\sqrt{2S_{ij}S_{ij}} \tag{6}$$

where S_{ij} is the strain-rate tensor:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{7}$$

The main drawback of this model is the specification of the mixing length which is different for every type of flow. Experience from previous Denton codes has shown that for turbomachinery applications, a limiter based on the blade pitch is appropriate:

$$l_{mix} = \begin{cases} \kappa y_n, & y_n < x_{lim} \\ \kappa x_{lim}, & y_n > x_{lim} \end{cases} \tag{8}$$

where κ is a constant, y_n is the normal distance from the nearest wall and x_{lim} is usually taken to be three percent of the pitch. y_n is calculated by Turbostream before the start of the main timestepping loop using the Poisson equation approach described by Tucker et al. [13].

To avoid having to use many grid points in the boundary layer, the flow is allowed to slip at the walls and a wall-function is used to obtain an expression for the wall shear stress. In this approach, it is assumed that the first grid point away from the wall lies either in the viscous sub-layer or in the logarithmic region of a turbulent boundary layer. In the former case the wall shear stress is approximated by

$$C_{f,w} = \frac{1}{Re_w} \tag{9}$$

and in the latter case by a curve fit to the log-law in the form of

$$C_{f,w} = -0.001767 + \frac{0.03177}{\ln Re_w} + \frac{0.25614}{(\ln Re_w)^2} \tag{10}$$

where $C_{f,w}$ is the coefficient of friction defined as

$$C_{f,w} = \frac{\tau_w}{\frac{1}{2}\rho U_w^2} \tag{11}$$

and Re_w is the cell Reynolds number defined as

$$Re_w = \frac{\rho U_w y_w}{\mu} \tag{12}$$

In the above equations, U_w is the velocity at the first grid node off the wall and y_w the height of the cell normal to the wall.

Convergence acceleration

To accelerate the convergence rate, Turbostream uses both spatially varying timesteps and a multi-grid scheme. In the former method, the timestep in each cell is limited by the local flow properties and geometry, allowing much larger timesteps to be used in the large free-stream cells that would otherwise be limited by the small cells in the boundary layer. The latter method uses multiple grid levels, each coarser than the preceding one, to accelerate the convergence by dispersing transients quickly on the coarser levels while retaining the spatial accuracy of the finest. In the Denton formulation used by Turbostream, adjacent cells are combined to form a grid of larger cells or blocks. The new coarse mesh is treated in just the same way as the original fine grid and so much larger timesteps are possible. The change in the value of **U** during one iteration is given by

$$\Delta \mathbf{U} = \left| \frac{\Sigma \mathbf{F}}{V} + \mathbf{S} \right|_{cell} \Delta t_{cell} + \sum_{blks} \left[\left| \frac{\Sigma \mathbf{F}}{V} + \mathbf{S} \right|_{block} \Delta t_{block} \right] \quad (13)$$

where the summation is over all the blocks to which the relevant cell belongs. Typically, three levels of multi-grid with a coarsening ratio of two are used.

Multistage and unsteady simulations

For steady-state calculations, multistage simulations are made possible by circumferentially averaging the flow entering a blade row, see Denton and Singh [14], Dawes [15], Denton [11]. The technique is applied at a mixing plane, the position of which is arbitrarily chosen by the user. The non-uniform flow upstream of this plane is mixed out so that it becomes pitchwise, but not spanwise, uniform downstream of the mixing plane. The process is conservative, but, like any mixing process, it is irreversible (see Fritsch and Giles [16]). In addition, care must be taken to avoid locating the mixing plane too close to leading or trailing edges, and thereby enforcing a non-physical circumferentially uniform flow.

For unsteady simulations, Turbostream implements the 'dual time stepping' technique proposed by Jameson [17]. This procedure allows the convergence acceleration methods described earlier to be used in time-accurate simulations by splitting the calculation up into a number of implicit 'outer loops' that iterate forwards in real time, each of which is comprised of a number of explicit 'inner loops' that converge the flow to a steady state in pseudo-time. Multistage unsteady simulations are enabled through an interpolation procedure that transfers information across a sliding interface that connects the upstream and downstream grid blocks which rotate relative to each other.

IMPLEMENTATION

During the initial considerations of implementing a flow solver to run on many-core architectures, there was some concern over the great number of different processors and programming models available. For GPUs alone, there are several options; these include the two main chip manufacturers AMD and NVIDIA and at least half a dozen programming languages and libraries that in some cases only target one vendor's GPUs. In this situation, it seems a daunting prospect to pick one combination that will have the longevity required for CFD solvers whose lifetime is often measured in decades.

For this reason, it was decided that another approach than a direct implementation was needed. Turbostream is therefore expressed as a series of subroutine definitions in the high-level Python scripting language. These definitions contain the input and output arguments for each subroutine, as well as the computations that are carried out within them. An as yet unpublished source-to-source compiler, which was developed for this work by the authors, is then used to transform these definitions into source code that can be further compiled for the target architecture that we wish to run on. Currently, the compiler can produce code for either multi-core CPUs or NVIDIA GPUs, with support for the Cell processor currently being developed. Aside from enabling the solver to run on many different processors from the same source code definition, this source-to-source compilation approach has two other main benefits:

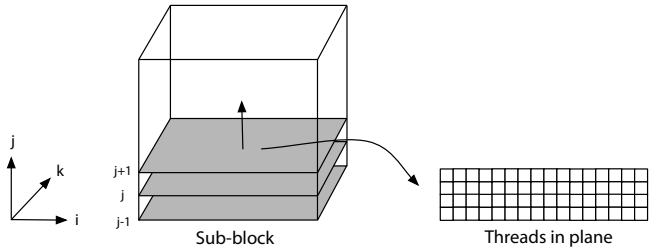

Figure 2. Iteration procedure for stencil subroutines

1. Since the definitions of the solver subroutines are completely separate from the source code that is actually produced for the target platform, the compiler is free to perform many different optimisations that would otherwise have complicated the code to an unacceptable degree. For an indication of the range and complexity of the many optimisations, of which only a subset are currently used by our compiler, necessary to achieve near-optimal performance on modern many-core architectures, see Datta et al. [18].
2. The use of a high-level language to express the logic of the solver makes it easier for domain scientists to add extensions such as new numerical schemes and turbulence models. This capability is important because the solver is intended to be a platform for academic research as well as a production code for day-to-day turbomachinery design.

In addition to the compiler, a runtime library has also been developed that takes care of memory management, subroutine invocation, file I/O and MPI communication. Since the details of the former two tasks are different for each processor, these parts of the library have to be written separately for each processor.

It should be noted that the source-to-source compilation strategy described here is only possible because of the limited range of computations that are performed by structured grid solvers. In short, each subroutine is a combination of stencil operations that use the nearest neighbours of a node to update its properties. The only computations that break with this paradigm are in the multigrid routine, which therefore has to be implemented seperately for each processor.

The main difficulty in producing efficient code for the stencil subroutines is in parallelising the computations across the hundreds of scalar processing units present on modern GPUs. The strategy used by the compiler is to split each grid block into smaller sub-blocks that are computed independently of eachother. One CUDA thread is started for each node in a plane of the sub-block. At the start of the subroutine, each thread loads in the necessary grid node values corresponding to its location. The threads then iterate upwards in the sub-block (Fig. 2), each time fetching a new plane and computing the values for the current one. Given that the shared memory can hold 16 KB, a typical sub-block size is 16x10x5. For such sub-block sizes, the surface-to-volume ratio is low enough to get good data reuse. The overall approach is similar to that described by Williams et al. [19] for structured grid applications on the Cell processor. A more detailed explanation of this implementation strategy and how it differs from that commonly used on CPUs has been included in Appendix A.

A final issue that has to be considered for a GPU implementation is that of data transfers across the PCI-Express bus which bridges the CPU and GPU memory spaces. The PCI-Express bus has a theoretical maximum bandwidth of 4 or 8 GB/s depending on whether it is of generation 1 or 2. When this number is compared to the bandwidth between the GPU's on-board GDDR3 memory and the GPU multi-processors (up to 141.7 GB/s), it becomes clear that any algorithm that requires a large amount of continous data transfer between the CPU and GPU will not achieve good performance. For a CFD solver, the obvious solution is to limit the size of the domain that can be calculated so that all of the necessary data can be stored in the GPU's on-board memory. Using this approach, it is only necessary to perform large transfers across the PCI-Express bus at the start of the calculation (the geometry) and at the end (the final flow solution). High-end GPUs today have up to 4 GB of on-board memory, sufficient to store all the data needed by Turbostream for a grid with $12 \cdot 10^6$ nodes, so this restriction is not a a significant limitation.

When operating in parallel across multiple GPUs, some boundary information must inevitably be transferred across the PCI-Express bus at the end of every time step. However, as will be shown in the next section, the low surface-to-volume ratios in turbomachinery grids mean that this data transfer is not a bottleneck.

PERFORMANCE

For any CFD solver, there are two important performance metrics:

1. How fast is the solver on a single processor?
2. How well does the performance of the solver scale when multiple processors are used together to tackle larger problems?

Table 1. Single processor performance

Solver	Processor	Time/node/step
TBLOCK	Intel Xeon 2.33 GHz	$5.1 \cdot 10^{-7}$ s
Turbostream	NVIDIA GT200	$2.7 \cdot 10^{-8}$ s

Turbostream dramatically increases the single-processor performance as compared to other solvers by using NVIDIA GPUs instead of traditional CPUs. To demonstrate this speed-up, we compare TBLOCK running on all four cores of an Intel Xeon 2.33 GHz CPU with Turbostream running on an NVIDIA GT200 GPU. TBLOCK was compiled with the Intel 10.1 Fortran compilers with automatic vectorisation and the highest degree of optimisation turned on, while Turbostream was compiled with NVIDIA's GPU compiler. Both solvers show approximately constant performance for grids with more than 10^5 nodes so a representative case with 10^6 nodes was used. The results are summarised in Table 1, which shows the time taken per grid node per timestep for each solver. The high performance of Turbostream running on the NVIDIA GPU is primarily due to the GPU's higher memory bandwidth, as well as the extra optimisations allowed by the source-to-source compilation. In terms of wall-clock time, both solvers converge in approximately the same number of timesteps, with a typical 300,000 node single-row calculation that requires 5,000 steps taking approximately 1 minute in total with Turbostream and 20 minutes in total with TBLOCK.

Over the last 10 years, CFD has become increasingly reliant on clusters of processors to enable more detailed simulations within design time frames. For this reason, the scalability of a solver across multiple processors can be equally important as its single-processor performance. A potential problem with increasing the single-processor performance by an order of magnitude is then that the multi-processor performance suffers since the time required to exchange boundary information remains roughly constant. However, the low surface-to-volume ratios in turbomachinery grids mean that good scalability can be achieved even with very fast solvers. To demonstrate this point, Fig. 3 shows the performance of Turbostream across a cluster of 16 NVIDIA G80 GPUs. There are four nodes in the cluster, each consisting of a traditional 1U server with a quad-core CPU connected through PCI-Express cables to another 1U server with four GPUs. The nodes are networked together with 1 Gigabit Ethernet interconnects.

To simplify the mesh generation for an arbitrary number of processors, we use an idealised case of simple flow through a square channel. In an attempt to represent a typical multi-stage turbomachinery calculation with one stage per GPU, the ratio

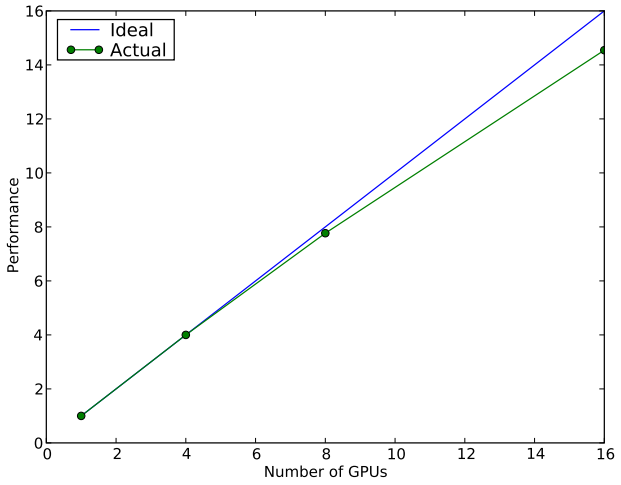

Figure 3. Turbostream weak scaling over multiple GPUs. Performance is measured as the inverse of the time per grid node per timestep.

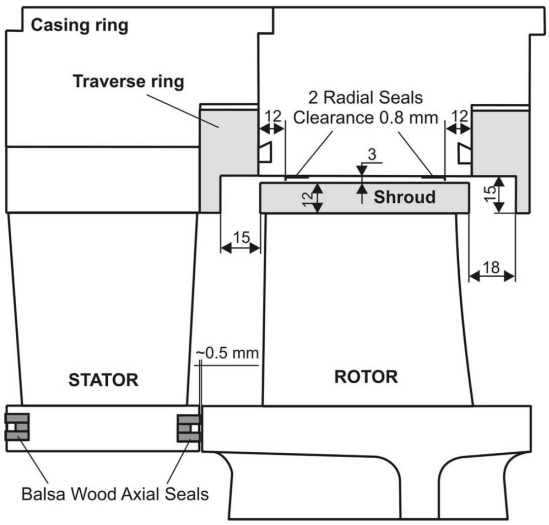

Figure 4. Single stage geometry

between the number of points in the axial, radial and circumferential directions is taken to be 4:1:1. Two million nodes are used per GPU, so the total size of the simulation scales with the number of GPUs used. In the authors' experience, this setup closely resembles real world usage – due to the high single-processor performance of the solver, multiple GPUs are only used in practice if the simulation is too large to fit in a single GPU's on-board memory.

As can be seen in Fig. 3, almost ideal scaling is obtained for 16 GPUs. It should also be noted that the interconnect used here (Gigabit Ethernet) has poor performance compared to other options, and that using a higher performance interconnect such as Infiniband should improve the scaling performance further.

VALIDATION

Validation is the biggest hurdle for any new flow solver to gain acceptance in the community. The authors are currently running through many different existing TBLOCK test cases with Turbostream. Although minor differences between the implementation of the two solvers mean that the results are not always identical, they are in all cases in close agreement with each other. A calculation of a three-stage turbine with leakage paths is presented in this work.

Three-stage turbine with leakage paths

The test case is a three-stage turbine with leakage paths. It was originally presented by Rosic et al. [7] to demonstrate the importance of shroud leakage modeling in multistage turbine flow calculations. The original work was carried out using TBLOCK. Here, we show that Turbostream is capable of producing similar results.

The arrangement of a single stage is is presented in Fig. 4, showing both the hub and shroud geometries. All leakage paths are fully represented in the CFD mesh, and a cavity with rotating walls has been added to represent the area below the hub. This stage is replicated three times to form the whole machine, resulting in an overall computational domain as shown in Fig. 5. The mesh used is of the H-type, and the total number of grid nodes is $4.5 \cdot 10^6$. A cluster with four NVIDIA G80 GPUs was used to calculate the flow, resulting in an overall run time of less than 10 minutes. Further computational and experimental details are given in the original paper.

Experimental and numerical results are presented using span-wise distributions of the pitchwise averaged exit yaw angle for the third stator and rotor, as well as exit total pressure coefficient contours for the third stator. The total pressure coefficient was obtained by nondimensionalizing the total pressure by the total pressure drop across the whole machine:

$$C_{p_0} = \frac{p_{0_{in}} - p_0}{p_{0_{in}} - p_{0_{ex}}} \tag{14}$$

Two sets of Turbostream results are presented; one with leakage flows and one without. For comparison, TBLOCK results for the case with leakages are also shown.

As should be expected, the results are similar to those of the original work (Rosic et al. [7]) and so only a brief discussion is warranted here. Fig. 6 shows a meridional view of pitch-

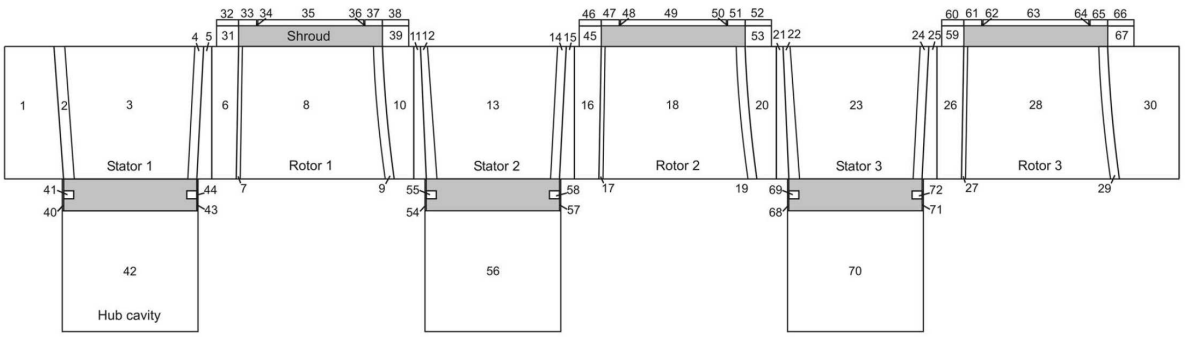

Figure 5. Computational domain

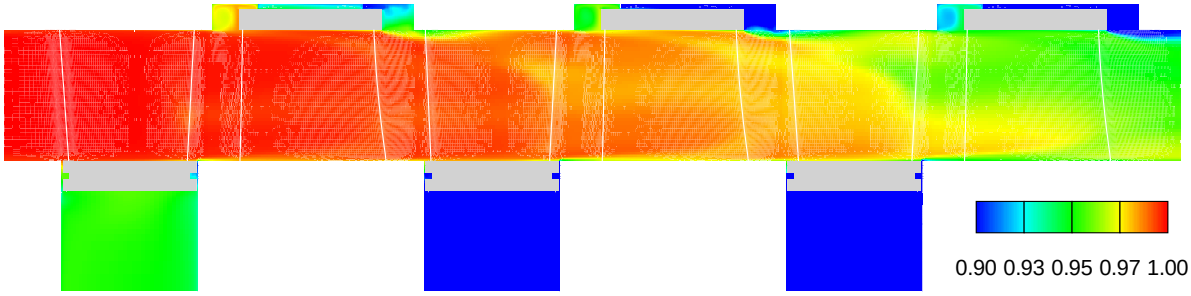

Figure 6. Pitchwise averaged entropy function: $\exp(-\Delta s/R)$ (Turbostream)

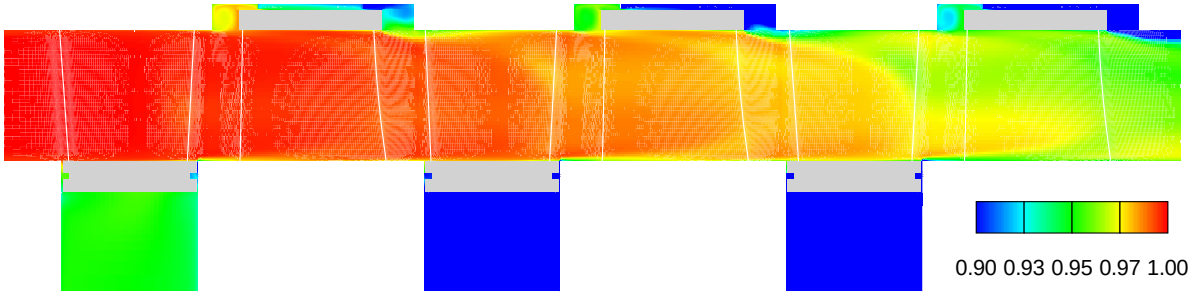

Figure 7. Pitchwise averaged entropy function: $\exp(-\Delta s/R)$ (TBLOCK)

wise averaged entropy function for Turbostream (Fig. 7 shows the same for TBLOCK). It is clear that the rotor shroud leakage flows interact strongly, and enhance the casing secondary flow in the following stator. Similarly, but to a lesser extent, the stator hub leakages add to the strength of the hub secondary flow in the following rotor. As an example of this effect, Fig. 8 shows a comparison of measured (using a 5-hole pneumatic probe) and calculated total pressure loss coefficient at the exit of stator 3. The experimental results show a dominant casing secondary flow loss core that has migrated to 50% span, and a smaller hub loss core at 25% span. With no leakage flows (clean hub and casing annulus lines), Turbostream predicts two distinct small loss cores at 15% and 85% span. With the addition of leakage paths, the agreement is much closer. In particular, the shroud leakage from rotor 2 has strengthened the stator 3 casing secondary flow and pushed the associated loss core toward mid-span. The remaining discrepancy between the CFD and experiment is likely to be largely the result of difficulties in obtaining the precise leakage gaps in the experiment (particularly at the hub). Finally, Fig. 9 compares the exit yaw angle distributions downstream of stator 3 and rotor 3. Again, the addition of shroud leakage improves the predictions of the stator 3 flow near the casing but, in this case, the accuracy of the rotor exit prediction has not been significantly improved by the inclusion of leakage paths.

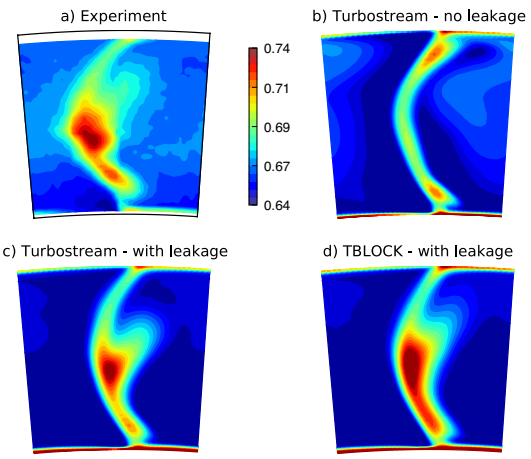

Figure 8. C_{p_0} contours - Stator 3

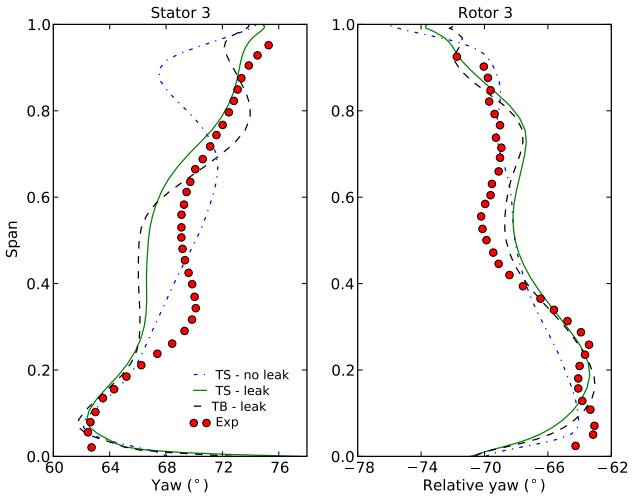

Figure 9. Measured and predicted pitchwise averaged yaw angle

DISCUSSION

The work presented here has shown that GPUs have enabled a dramatic acceleration of Turbostream (19 times speed-up on a single GPU versus a quad core CPU) as compared to the original Fortran solver, TBLOCK. Such a step-change will have two clear implications for the turbomachinery design process:

First, as demonstrated by the three-stage turbine calculation presented in this paper, it is now possible to perform steady-state simulations of whole machines in less than 10 minutes, even on clusters of moderate size and cost. Furthermore, single blade calculations are approaching interactive time scales on desktop computers with a single GPU.

Second, the performance offered by Turbostream enables the use of high-fidelity methods in the design process. For example, full annulus unsteady simulations, which are not currently routine for design work, can now be done in calculations that can be left to complete overnight.

CONCLUSIONS

The main conclusion that can be drawn from this work is that massively parallel architectures such as GPUs can provide an order of magnitude greater performance than traditional CPUs for CFD solvers. However, taking advantage of processors such as the GPU requires a complete rewrite of the solver. We argue that the rapidly changing many-core processor landscape means that the use of a source-to-source compiler to decouple the solver's definition from its implementation is crucial. This approach has the added benefit of allowing for the use of complicated optimisation strategies that would otherwise make it difficult for CFD developers to recognise the underlying algorithm in the source code.

For turbomachinery design, the dramatic increase in performance offered by Turbostream will open up new levels of interactivity in three-dimensional design, as well as enabling the use of high-fidelity methods in the routine design process.

ACKNOWLEDGMENT

The authors would like to thank NVIDIA for donating the GPU hardware used in this work. In addition, the authors are grateful to Budimir Rosic of the Whittle Laboratory for providing the three-stage turbine test case.

REFERENCES

[1] Brandvik, T., and Pullan, G., 2007. "Acceleration of a two-dimensional Euler solver using commodity graphics hardware". *IMechE Journal of Mechanical Engineering Science*, **221**(12), pp. 1745–1748.

[2] Brandvik, T., and Pullan, G., 2008. "Acceleration of a 3D Euler Solver using Commodity Graphics Hardware". *46th AIAA Aerospace Sciences Meeting, Reno, NV*.

[3] Elsen, E., LeGresley, P., and Darve, E., 2008. "Large calculation of the flow over a hypersonic vehicle using a GPU". *J. Comput. Phys.*, **227**(24), pp. 10148–10161.

[4] Borkar, S., 2007. "Thousand core chips: a technology perspective". In DAC '07: Proceedings of the 44th annual conference on Design automation.

[5] Klostermeier, C., 2008. "Investigation into the Capability of Large Eddy Simulation for Turbomachinery Design". PhD thesis, University of Cambridge.

[6] Reid, K., Denton, J., Pullan, G., Curtis, E., and Longley, J., 2007. "The Interaction of Turbine Inter-Platform Leakage Flow With the Mainstream Flow". *ASME J. Turb.*, **129**(2), pp. 303–310.
[7] Rosic, B., Denton, J. D., and Pullan, G., 2006. "The Importance of Shroud Leakage Modeling in Multistage Turbine Flow Calculations". *ASME J. Turb.*, **128**(4), pp. 699–707.
[8] Denton, J. D., 1975. "A Time Marching Method for Two and Three Dimensional Blade to Blade Flow". *Aero. Res. Coun. 3775*.
[9] Denton, J. D., 1982. "An Improved Time Marching Method for Turbomachinery Flow Calculation". *ASME 82-GTP-239*.
[10] Denton, J. D., 1990. "The Use of a Distributed Body Force to Simulate Viscous Effects in 3D Flow Calculations". *ASME 86-GT-144*.
[11] Denton, J. D., 1990. "The Calculation of Three Dimensional Viscous Flow through Multistage Turbomachines". *ASME 90-GT-19*.
[12] Denton, J. D., 2002. "The effects of lean and sweep on transonic fan performance". *TASK Quart.*, pp. 7–23.
[13] Tucker, P. G., Rumsey, C. L., Spalart, P. R., Bartels, R. E., and Biedron, R. T., 2005. "Computations of Wall Distances Based on Differential Equations". *AIAA Journal*, **43**.
[14] Denton, J. D., and Singh, U. K., 1979. "Time Marching Methods for Turbomachinery Flow Calculation". *VKI Lecture Series - 1979-7*.
[15] Dawes, W. N., 1992. "Toward Improved Throughflow Capability: The Use of Three-Dimensional Viscous Flow Solvers in a Multistage Environment". *ASME J. Turb.*, **114**(8), pp. 8–17.
[16] Fritsch, G., and Giles, M. B., 1992. "Second-order Effects of Unsteadiness on the Performance of Turbomachines". *ASME 92-GT-389*.
[17] Jameson, A., 1991. "Time Dependent Calculations Using Multigrid, with Applications to Unsteady Flows Past Airfoils and Wings". *AIAA 91-1596*.
[18] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K., 2009. "Stencil Computation Optimization and Autotuning on State-of-the-Art Multicore Architectures". In Supercomputing 2009.
[19] Williams, S., Carter, J., Oliker, L., Shalf, J., and Yelick, K., 2007. "Scientific Computing Kernels on the Cell Processor". *International Journal of Parallel Programming*, **35**(3), pp. 263–298.

Appendix A: Implementation details

To illustrate how the implementation of stencil operations differs for CPUs and GPUs, we will consider the simple second order smoothing stencil defined below:

$$b_{i,j,k} = (1-s)a_{i,j,k} + \frac{s}{6}(a_{i-1,j,k} + a_{i+1,j,k} + a_{i,j-1,k} + a_{i,j+1,k} + a_{i,j,k-1} + a_{i,j,k+1}), \quad (15)$$

where a and b are values in a structured grid indexed by i, j, k, and s is a factor controlling the amount of smoothing.

To simplify the problem, we only consider a computational domain consisting of a single block. The block has the dimensions NI-2, NJ-2 and NK-2 in the three coordinate directions. In memory, this block is represented as a three-dimensional array with dimensions NI, NJ and NK, where the extra two points in each dimension contain ghost cells, one on either side of the domain in each dimension. These ghost cells are assumed to contain the appropriate values so that whatever boundary conditions exist around the block are satisfied when we perform the stencil operation at the edges of the domain.

Listings 1 and 2 contain examples of the implementation of the stencil for a CPU and an NVIDIA GPU respectively. The CPU implementation is in Fortran 77; the GPU implementation in NVDIA's CUDA language. The examples include the memory allocation, the calling of the subroutine and the definition of the subroutine itself (note that subroutines are referred to as kernels on the GPU). For the sake of brevity, we do not show the initialization of the memory. The CPU implementation should be familiar to most CFD developers. It consists of a simple nested loop over the computational domain, with the inner computation performing the stencil operation.

The GPU implementation is more complicated. First, there are now two disjoint memory spaces to manage, one for the CPU and one for the GPU. It is therefore necessary to allocate memory on both the CPU (line 5) and the GPU (lines 8 and 9), and then transfer data from the CPU to the GPU (line 12). Any operations that involve GPU memory outside of a kernel require calls to special functions implemented by the NVIDIA GPU driver, these have the prefix *cuda*. Second, since a GPU kernel is executed in parallel by many threads at the same time, it is necessary when calling it to specify how many threads are needed and how these should be organised. Here, we assume that the domain is small enough to fit in a single sub-block, so that only a single "thread block" is needed (line 15). A thread block is a CUDA term for a group of threads that operate together, and are executed in parallel on the same multi-processor. In this particular example, the thread block consists of a single plane of threads (line 16). Note that CUDA's facility for multi-dimensional thread blocks is used to simplify the indexing in the kernel later. Finally, the kernel is called with the the required number of threads (line 19).

Listing 1. Fortran implementation for a CPU

```
      REAL A(NI,NJ,NK), B(NI,NJ,NK), SF

      CALL SMOOTH(SF, A, B)

      SUBROUTINE SMOOTH(SF, A, B)

C     LOOP OVER DOMAIN
      DO K=2,NK-1
      DO J=2,NJ-1
      DO I=2,NI-1
      B(I,J,K) = (1.0-SF)*A(I+1,J,K) +
     &           SF*(A(I-1,J,K) + A(I+1,J,K) +
     &               A(I,J-1,K) + A(I,J+1,K) +
     &               A(I,J,K-1) + A(I,J,K+1))/6.0
      END DO
      END DO
      END DO
      RETURN
      END
```

Regarding the implementation of the kernel itself, the following points should be noted:

1. Two CUDA-specific keywords are used. The GPU kernel is defined as **__global__** (line 22), which means that it is called from the CPU and is exectuted on the GPU. The array storage in the kernel is defined as **__shared__** (line 27), which means that the arrays are stored in the 16KB on-chip memory associated with each of the GPU's multi-processors.
2. Each thread uses the built-in variable **threadIdx** to find its i and k coordinate in the plane of a sub-block (lines 31 and 32).
3. The variables **jm1**, **j** and **jp1** are used to hold the offsets to the $j-1$, j and $j+1$ planes in shared memory (line 38) . These are cycled at the end of each iteration (line 58) so that the new plane that is loaded during the next iteration replaces the one that is no longer required by the stencil operation.
4. Data from the array **a_d** in the GPU's main memory is explicitly loaded into the array **a** in shared memory (lines 34, 35 and 44). The threads in the plane load one value each, so the built-in function **__syncthreads()** has to be called to make sure all the threads have finished loading data before progressing further in the code.
5. The outer threads only load data from the ghost zones and do not participate in the computation with the inner threads (line 49).

Listing 2. CUDA implementation for an NVIDIA GPU

```
1  /* Macro for 3D to 1D index translation */
2  #define I3D(ni,nj,i,j,k) ((i)+(ni)*(j)+(ni)*(nj)*(k))
3
4  float *a_cpu, *a_gpu, *b_gpu, sf;
5
6  /* allocate memory on the host (CPU) */
7  nbyte = sizeof(float)*NI*NJ*NK;
8  a_h = malloc(nbyte);
9  /* allocate memory on the device (GPU) */
10 cudaMalloc(&a_d, nbyte);
11 cudaMalloc(&b_d, nbyte);
12
13 /* transfer memory from host to device */
14 cudaMemcpy(a_d, a_h, nbyte, cudaMemcpyHostToDevice);
15
16 /* GPU kernel parameters */
17 num_threadblocks = dim3(1,1,1); // single thread block
18 num_threads = dim3(NI,NK,1); // plane of threads
19 /* invoke GPU kernel */
20 smooth_kernel<<<num_threadblocks, num_threads>>>(
21 a_d, b_d, sf);
22
23 __global__ void smooth_kernel(
24   float sf, float *a_data, float *b_data)
25 {
26   int i, j, jm1, jp1, k, j_plane;
27   /* shared memory for three planes */
28   __shared__ float a[NI][3][NK];
29
30   /* current thread index */
31   i = (int)threadIdx.x;
32   k = (int)threadIdx.y;
33
34   /* fetch the first planes into shared memory */
35   a[i][0][k] = a_d[I3D(NI, NJ, i, 0, k)]
36   a[i][1][k] = a_d[I3D(NI, NJ, i, 1, k)];
37
38   /* set initial jm1, j, jp1 */
39   jm1 = 0;  j = 1;  jp1 = 2;
40
41   /* iterate upwards in j-direction */
42   for (j_plane=1; j_plane < NJ-1; j_plane++) {
43
44     /* read the next plane into the jp1 slot */
45     a[i][jp1][k] = a_d[I3D(NI, NJ, i, j_plane+1, k)]
46     /* make sure reads into shared memory are done */
47     __syncthreads();
48
49     /* ghost-zone threads don't compute */
50     if (i > 0 && i < ni-1 && k > 0 && k < nk-1) {
51       /* apply stencil and write out result */
52       i000 = I3D(NI, NJ, i, j, k);
53       b_d[i000] = (1.0f-sf)*a[i][j][k] +
54         sf*(a[i-1][j][k] + a[i+1][j][k] +
55             a[i][jm1][k] + a[i][jp1][k] +
56             a[i][j][k-1] + a[i][j][k+1])/6.0f;
57     }
58     /* cycle j indices */
59     tmp = jm1; jm1 = j; j = jp1; jp1 = tmp;
60   }
61 }
```